Programação Imperativa

Licenciatura em Ciências da Computação

Universidade do Minho

JBA AT di.uminho.pt

2015/2016

Pesquisa e Ordenação

1. Pesquisa em vectores
   - Pesquisa Sequencial
   - Pesquisa Binária
2. Ordenação de Vectores
   - Ordenação por inserção
   - Ordenação por selecção
   - Estratégia "divide-and-conquer"
     - `merge-sort`
     - `quick-sort`

Pesquisa num vector

- Os arrays possibilitam um acesso muito eficiente a uma qualquer posição (índice)
- ...mas muitas vezes pretende-se procurar um valor num array (sem se saber a posição que ocupa).
- Esse problema é designado por **pesquisa num array**. Pode ser expresso da seguinte forma:

  Dado um array a (com n elementos) e um valor x, determinar em que posição ocorre x em a (retornando -1 se x não ocorrer em a).

Pesquisa sequencial

- Não existindo informação adicional sobre a forma como estão distribuídos os elementos no array (e.g. se estão organizados de forma ordenada), só nos resta a alternativa de "percorrer todo o array até encontrar o elemento pretendido".
- Esse algoritmo é habitualmente designado por **pesquisa sequencial** (ou linear).
- Se existirem garantias que o array se encontra ordenado, é possível adaptar o algoritmo por forma antecipar a saída em certos casos (quando detecta um elemento maior que o pretendido).

Pesquisa Binária (em vectores ordenados)

- Quando existem garantias que o array está ordenado, é possível realizar a pesquisa de um elemento num vector de forma significativamente mais eficiente: a **pesquisa binária**
- A estratégia consiste em replicar o procedimento normalmente adoptado quando se pesquisa uma palavra num dicionário:
  - considera-se o valor da posição intermédia do array:
    1. se esse valor for o elemento pretendido, retorna-se a respectiva posição;
    2. se for maior, repete-se o procedimento considerando unicamente a primeira metade do vector;
    3. se for menor, repete-se o procedimento considerando unicamente a segunda metade.
- Este algoritmo pode ser traduzido directamente em C através da seguinte função recursiva

```
int binsearchR(int a[], int x, int l, int h) {
  int m = (l+h)/2;
  if (h<l) return -1;
  if (a[m]==x) return m;
  if (x < a[m]) return binsearchR(a,x,l,m);
  else return binsearchR(a,x,m+1,h);
}
```

- Em C, será por ventura mais natural codificar a pesquisa binária de forma iterativa:

```
/*
   pesquisa binaria do elemento "x" no array "a" (com tamanho "n")
*/
int binsearch(int a[], int x, int n) {
  int l=0, h=n-1, m, idx=-1;
  /* obs: "l" e "h" denotam respectivamente os indices minimos e maximos
     da ''janela'' do array que ainda falta explorar                    */
  while (idx<0 && l<=h) {
    m = (l+h)/2;
    if (a[m]==x) {
      idx=m;
    } else if (x < a[m]) {
      h = m-1;
    } else {
      l = m+1;
    }
  }
  return idx;
}
```

Inserção Ordenada

- Uma forma de se garantir que um array está ordenado consiste em estabelecer essa garantia **por construção** (i.e. a forma como o array é construído garante que se encontra sempre ordenado)
- Para isso, interessa considerar o problema de **inserção ordenada**

  Dado um array ordenado a (com n elementos), inserir um novo elemento x nesse array mantendo a propriedade de ordenação

- Para realizar uma inserção ordenado torna-se necessário:
  1. Determinar a posição do array onde o novo elemento deve ser inserido;
  2. Arranjar espaço para essa inserção (deslocando todos os contidos no array a partir dessa posição uma posição para a frente)
- Uma forma expedita de realizar essas operações consiste em percorrer o array do fim para o início deslocando os elementos numa posição, até que se encontre a posição onde deverá ser adicionado o elemento.

- O algoritmo descrito pode ser directamente codificado como uma função recursiva:

```
/*
 Insere ordenadamente um elemento "x" num array "a" (de tamanho "n")
 Retorna o numero de elementos do array resultante
*/
int insert(int a[], int x, int n) {
  int i;
  for (i=n; i>0 && a[i-1]>x; i--) {
    a[i] = a[i-1];
    }
  a[i] = x;
  return n+1;
}
```

Ordenação de um vector

- Dado um array não ordenado, podemos ter interesse um torna-lo ordenado (mantendo os mesmos elementos, apenas trocando as respectivas posições)
- Problema de **ordenação de um array**: dado um array não ordenado `a` (com `n` elementos), trocar as posições onde ocorrem os vários valores por forma a que o array resultante
  1. contenha exactamente os mesmos elementos
  2. esteja ordenado

Ordenação por inserção (*insertion-sort*)

- Uma forma muito simples para se ordenar um array consiste em inserir ordenadamente todos os elementos de um array num novo array.
- Para mais, essa operação pode ser realizada *in-place* (i.e. sem ter necessidade de alocar um novo array) — a ideia é que à medida que vamos percorrendo o array original, arranja-se espaço para realizar a inserção ordenada.

```
/*
  insertion-sort
*/
void insertionSort(int a[], int n) {
  int i;
  for (i=1; i<n; i++) {
    insert(a, a[i], i);
  }
}
```

insertion-sort (cont.)

- Alternativamente, numa única função...

```
/*
  insertion-sort
*/
void insertionSort(int a[], int n) {
  int i, j, x;
  for (i=1; i<n; i++) {
    x = a[i];
    for (j=i; j>0 && a[j-1]>x; j--) {
      a[j] = a[j-1];
    }
    a[j] = x;
  }
}
```

Ordenação por selecção (e.g. *max-sort*)

- Uma estratégia alternativa para proceder a ordenação de um vector consiste em seleccionar o elemento do vector original que irá ocupar uma posição específica no vector ordenado resultante.
  - O primeiro elemento no vector ordenado deverá ser o mínimo do vector original;
  - O último elemento do vector ordenado deverá ser o máximo do vector original.
- Uma vez identificado esse elemento, realiza-se a sua troca com o que ocupa posição correcta (e procede-se de igual forma para os restantes elementos)

```
/* swap de dois elementos de um array */
void swap(int a[], int i, int j) {
  int temp = a[i];
  a[i] = a[j]; a[j] = temp;
}

/* max de um array (-1 se nao existirem elementos) */
int maxArray(int a[], int n) {
  int imax=n-1, i;
  for (i=0; i<n-1; i++)
    if (a[i]>a[imax]) imax=i;
  return imax;
}

/* max-sort */
void maxSort(int a[], int n) {
  int i;
  for (i=n-1; i>=0; i++)
    swap(a, i, maxArray(a,i+1));
}
```

Estratégia *divide-and-conquer*

- Muitas vezes, consegue-se resolver um dado problema se admitirmos que conseguimos resolver o mesmo problema em instâncias mais pequenas
- E.g. no caso da ordenação: se admitirmos que conseguimos ordenar cada uma das duas metades de um array, podemos construir o array ordenado final combinando os elementos de cada uma dessas metades (operação de `merge` de dois arrays ordenados)
- Combinando com o facto que para problemas de dimensão muito pequena existe uma solução trivial (e.g. quando um array tem um único elemento está trivialmente ordenado), somos assim conduzido a um algoritmo recursivo para a estratégia **divide-and-conquer**:
  - Se a dimensão do problema admite uma solução trivial, retorna essa solução;
  - Senão:
    1. particiona-se o problema em n sub-problemas de dimensão mais pequena (normalmente apenas 2);
    2. aplica-se recursivamente o algoritmo a cada um desses sub-problemas;
    3. combinam-se os resultados dos sub-problemas por forma a resultar na solução do problema original

merge-sort

- Uma aplicação directa da estratégia *divide-and-conquer* é patente no algoritmo **merge-sort**
  1. O array é dividido em duas metades
  2. A cada uma delas é aplicado recursivamente o algoritmo até que a solução seja trivial (dimensão do array ≤ 1)
  3. Tendo ambas as metades ordenadas, constrói-se o resultado final combinando os elementos de ambas as listas já ordenadas (operação de `merge`)
- A operação de *merge* percorre simultaneamente cada um dos arrays ordenados, seleccionando a cada passo o menor dos dois elementos considerados.
  - no final, deverá ainda copiar os elementos que permanecem no array resultante

```
void merge(int v[], int a[], int na, int b[], int nb) {
  int i=0, j=0, k=0;
  while(i<na && j<nb)
    if (a[i]<=b[j]) v[k++] = a[i++];
    else v[k++] = b[j++];
  while (i<na)
    v[k++] = a[i++];
  while (j<nb)
    v[k++] = b[j++];
}
void msort (int v[], int n, int temp[]) {
  int i , m;
  if (n>1) {
    m = n/2;
    msort (v, m, temp);
    msort (v+m, n-m, temp);
    merge (temp, v, m, &(v[m]), n-m);
    for (i=0; (i<n); i++) v[i] = temp[i];
  }
}
void mergesort (int v[], int n) {
  int aux[n];
  msort(v, n, aux);
}
```

quick-sort

- Uma forma alternativa de aplicar a estratégia *divide-and-conquer* consiste no que se designa por algoritmo **quick-sort**
  1. Selecciona-se um qualquer elemento do array (*pivot*), dividindo-se os restantes elementos do array nos que são menores e maiores do que o *pivot* (operação `partition`)
  2. Aplica-se recursivamente o algoritmo até que a solução seja trivial (dimensão do array ≤ 1)
  3. Para compor o resultado final, basta colocar o *pivot* entre a parte do array com os elementos menores, e a parte com os elementos maiores.
- A operação `partition` pode ser realizada numa única passagem sobre o array.
- O comportamento do algoritmo é tanto melhor quanto mais equilibrada forem as dimensões dos sub-problemas gerados
  - Em particular, note-se que se o *pivot* for o máximo (ou mínimo) do array, um dos sub-problemas será o array vazio

```
int partition (int v[], int l, int h) {
  int i=l; //posicao do primeiro elemento maior que pivot
  int j=l; //posicao do primeiro elemento nao tratado
  // pivot: v[h] (ultimo elemento)
  while (j<h-1) {
    if (v[j]<v[h])
      swap(v, i++, j++);
    else
      j++;
  }
  swap(v, i, h); //coloca pivot entre as duas partes
  return i;
}
void qsortAux (int v[], int a, int b) {
  int p;
  if (a<b) {
    p = partition (v, a, b);
    qsortAux (v , a , p-1);
    qsortAux (v, p+1, b);
  }
}
void qsort (int v[] , int n){
  qsortAux (v,0 ,n-1);
}
```

Ordenação de vectores: conclusão

- Foram estudados diferentes algoritmos para a resolução do problema de **ordenação de um array**
- Esses algoritmos diferem na estratégia seguida para atingir o objectivos pretendido, exibindo características distintas em termos da sua **eficiência** (i.e. o tempo de execução)
- Os algoritmos baseados na estratégia "divide-and-conquer" são os que exibem melhores comportamentos em termos de eficiência.
- Nestes, note que ao contrário do `merge-sort` onde o maior esforço é colocado na fase de "agregar os resultados" (operação `merge`), no `quick-sort` o esforço é colocado na fase de "divisão nos sub-problemas" (operação `partition`).